Reasoning about the Semantic Web using Answer Set Programming

Giovambattista Ianni, Francesco Calimeri, Vincenzino Lio, Stefania Galizia, and Agata Bonfà

Department of Mathematics, Università della Calabria,
Via Pietro Bucci, 30B
87036 Rende (CS), Italy

Abstract. The paper discusses some innovative aspects related to the integration of a framework based on Answer Set Programming in an Information Retrieval Agent, namely, the Global Search Agent. In order to improve the original system effectiveness, the GSA_2 system introduces a new internal architecture based on a message-passing framework and on an ontology description formalism (WOLF, Web OntoLogy Framework), conceived in order to describe and make reasoning on "facts about the web and the user". The role of a central intelligence is played by a reasoning system based on Answer Set Programming; it makes the Agent able to take independent decisions. The high expressive power of Answer Set Programming allows to describe, program and plan behaviors of the Agent easily and quickly, and to experiment with any (even future) Information Retrieval strategy. Both the System Architecture and WOLF are very general and reusable, and constitute a very good and interesting example of actual exploiting of Answer Set Programming for real applications.

1 Introduction

Disjunctive logic programs are logic programs where disjunction is allowed in the heads of logical rules and negation may occur in the bodies of the rules. Such programs are now widely recognized as a valuable tool for knowledge representation and common-sense reasoning [6, 1]. One of the attractions of disjunctive logic programming (DLP) is its capability of allowing the natural modeling of incomplete knowledge. The most widely accepted semantics is the *answer sets semantics* proposed by Gelfond and Lifschitz [14] as an extension of the stable model semantics of normal logic programs [13]. According to this semantics, a disjunctive logic program may have several alternative models (but possibly none), called *answer sets*, each corresponding to a possible view of the world. The Answer Set Programming (ASP, in the following) community is nowadays moving towards actual applications of this semantics [25, **?**,29, 9, 26]. ASP points of strength are high expressiveness, declarativity, and easiness of program encoding. Nonetheless, ASP cannot be directly exploited in many contexts where 'decisions' have to be taken at very high paces, such as real time systems, and agent systems.

This paper constitutes a very first example about the effort we are carrying out in order to embed an ASP engine within an Agent architecture.

The Global Search Agent, illustrated in [16, 3], is a prototype conceived in order to improve web search quality on both the recall (the percentage of documents covered by a search engine) and the precision (the percentage of interesting documents covered by a search engine), and to assist the user through user profiling techniques. It is, in a sense, a very aggressive document collector. Much of the web retrieval techniques are exploited and combined in a single framework:

– Meta-Searching: in order to enhance coverage, a lot of search engines are queried in parallel [15, 17].
– Anticipated exploration: linked documents are explored and filtered in advance, independently from user's intervention [21].
– User profiling: the user is made able to express preference on selected documents [4, 20].
– Document ontology design: the user can classify his search in a hierarchical taxonomy. Such "concept tree" information is employed to tailor the anticipated exploration to a fine-tuned search space [19, 22].

The version of GSA herein presented (GSA_2 or the Agent, in the following), introduces an ASP reasoning module within the overall architecture: this allows a much better integration of the former modules, and allows the Designer to experiment, combine and evaluate any reasoning strategy in a quick, clear and declarative way.

Improvements and benefits of the new architecture can be resumed in the following points:

1. the internal structure of GSA_2 is conceived as a set of cooperating agents; each agent is a specialized module plugged within the system giving the Agent its (many) own capabilities; new functionalities might be easily added introducing new kinds of agents;
2. a new Peer to Peer module is introduced, in order to allow the Agent to exchange knowledge throughout a network of peer GSA_2 instances;
3. a Web Ontology Framework (WOLF in the following) is introduced. WOLF contains logic primitives intended to model the Web, the internal document ontology, the user(s) profile(s), the web information sources (like traditional search-engines), the document contents, and the interaction and knowledge exchange among peers;
4. the system is equipped with an internal (logic) reasoning engine. This allows the Agent to take its own decisions about the way to retrieve information, answer to user explicit and implicit requests, classify documents, and interact with peers.

This new architecture produces many benefits for the whole system, as well as for the Developer, the Designer, the End User. In the next section we briefly show details on how the system works, pointing out the internal multi-agent architecture, the message-passing functionalities, the knowledge representation framework, the see-through multiple information retrieval capabilities, the relationships with the user.

We show then in Section 3 the core of the KatheRinE Agent (the Knowledge & Reasoning Agent), its structure and the interactions with the rest of the system;

In Section 3.2 we present the Knowledge and Ontology Representation Framework, as well as its main purposes (modelling document ontologies, knowledge exchange, meta-search and user profiles). Some little pieces of code are then proposed in order to give an idea of the ease and the power of ASP in this context. Eventually, the section 4, quickly resumes the main aspects of the ideas and the actual system, discuss about the current status of the prototype and the already decided improvements, as well as future directions of the research.

2 System overview and architecture

The internal architecture is of GSA (Figure 1) is conceived as a multi-agent system, where many (neither predefined nor fixed) types of agents interact through a message passing strategy.

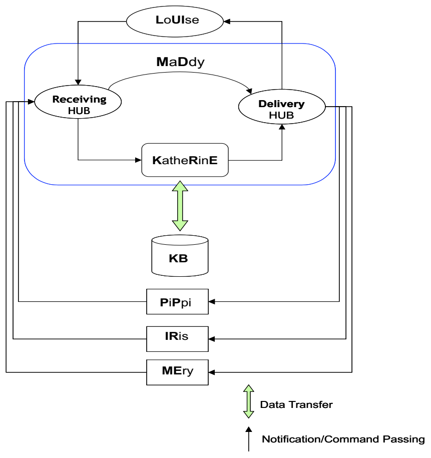

Fig. 1. Architecture of the GSA_2 system

Messages are of two kinds: “notifications” and “commands”. Notifications are sent out when an agent intends to signal interesting events to the community of agents. For instance, the loUIse agent (User Interface Agent), notifies the MaDdy agent ([Message Delivery Agent) of any user interaction with the system (e.g. clicks on any links, text typed in any field).

Commands represent requests of particular actions directed from agent to agent. For instance, when the KatheRinE agent decides that it is worth to suggest a given document to the user, it sends a suitable command message to loUIse.

Each kind of agent has its own vocabulary for notifications and commands it can receive or send. The notification and commands dictionary has been conceived taking in account the FIPA message format [12]; the final version of GSA_2 will be able to work using the JADE platform [10], in order to exploit an already well-know and tested reality.

The current GSA system architecture includes the following agents:

1. MaDdy. The Message Delivery Agent. Messages are forwarded from senders to receivers through MaDdy. This is not an agent in strict sense, since it acts as an almost passive message delivery service.
2. KatheRinE. The Knowledge & Reasoning Agent. It acts as a central intelligence module. KatheRinE builds up its own fact base capturing messages going through itself. Furthermore, KatheRinE is able to perform reasoning, take decisions, and deliver commands to other agents based upon its knowledge base. The internal architecture of KatheRinE is explained in detail in Section 3.
3. MEry. The Metasearch Engine Agent. It is equipped with a collection of wrappers able to query a given set of traditional search engines, throughout the world wide web. This module has been redesigned since its first version, and now it has the capability to (re)generate wrappers in a semi-automatic way interacting with the user. MEry can be asked for a general keyword search, or for a search on a specific search engine. As soon as they are available, MEry extracts sets of URLs from results set of the search engines. Such sets of URLs are then notified to the system.
4. IRis. The Information Retrieval Agent. It embeds most of the technology introduced with the first release of GSA. This module, on request, is able to: a) score given documents with respect to a set of relevant keywords, b) classify documents, c) extract words pertaining a given topic, d) explore the web graph on its own, searching for relevant documents.
5. PiPpi. The Peer to Peer Agent. It manages the connections of a single GSA_2 instance to a network of GSA_2 instances (the GSANetwork in the following), by means of a suitable P2P platform. Through PiPpi the system is made able to share knowledge from a system to another. In general, any kind of notification and command message can be sent and received to and from the GSA_2 Network. This way, the system becomes able to exchange knowledge about each single fact base.
6. loUIse. The User Interface Agent handles input and output to and from the final user. It is able to record any user activity as well as assisting the user during her retrieval activities.

3 The KatheRinE agent

As briefly introduced above, KatheRinE is the reasoning module of the GSA_2system. The role played by KatheRinE is central, since it basically *implements* the behavior of the system.

This module acts as intelligent supervisor-colleague for all other internal agents. KatheRinE knows almost all about what happens inside of the Agent (i.e., some results have been found, some documents have been scored, etc.) and outside of it (i.e., the user has performed some activity, the P2P network has been queried, etc.), and *a)* chooses what has to be remembered, *b)* decides if some reaction has to be performed, and which one.

GSA_2 adopts the Answer Set Programming (ASP, in the following[1]) as knowledge representation and reasoning framework, and the DLV system as internal “engine”.

[1] Name coined by Vladimir Lifschitz in the invited talk at ICLP’99, and widely preferred, lately, to Disjunctive Logic Programming, or DLP.

There are many points scored by ASP and DLV that led us to these choices: first of all, the formalism has a very high expressive power; as better exposed below, under answer sets semantics, disjunctive logic programs completely capture the complexity class Σ_2^P. But, more interesting, the formalism is really highly declarative, and encoded logic programs are usually very concise, simple, and quick to design. The two most widespread ASP systems are DLV [7] and Smodels [28]. Due to space limitations, we will assume the reader is familiar with ASP programming and the DLV system. A thorough definition of concepts herein adopted can be found in [7].

3.1 Embedding the reasoning module

In order to provide the whole system with the ASP reasoning capabilities, we introduced a mapping between the GSA message format and the answer set programming.

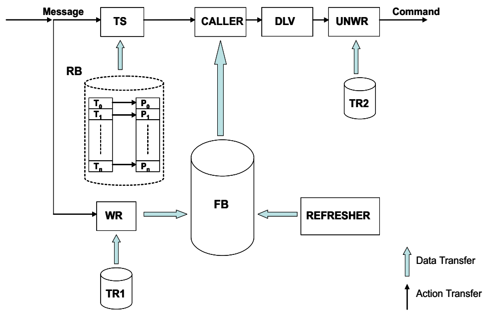

Fig. 2. Architecture of the KatheRinE agent

As it comes out from Figure 2, the KatheRinE agent is able to capture any message going through MaDdy, and to send out messages as well. In order to have a fully operating system from early stages of implementation, messages can bypass KatheRinE as well, allowing direct interaction among the other agents. When this “bypass” pipe is employed, GSA$_2$ behaves similarly to its former architecture. The behavior of KatheRinE is two-folded. First, it captures any message going through MaDdy. This way KatheRinE, populates its own fact base with knowledge about the past history of the system, e.g. queries made by the user, documents retrieved, topics classified, best performing external engines, “most reliable” external agents, etc. Second, it may react to special notification messages (triggers) and answer by providing command messages to other agents.

Figure 2 shows in detail how KatheRinE is conceived. An incoming message m is treated two ways:

1. A wrapper (WR) takes m and translates it in terms of a set of facts F_m. F_m is then transferred to the Fact Base (FB). This way any event is logged and available for reasoning on it. WR takes advantage of a translation table *TR1*, stating a mapping among logical facts and the GSA$_2$ messages.

2. The Reasoning Base (RB) consists of a set of couples $\{\langle t_0, p_0\rangle, \ldots, \langle t_n, p_n\rangle\}$, where each t_i is a message trigger type, and p_i is an associated logic program. A trigger selector (TS) looks up within RB and finds whether the current message trigger type appears in it. If this is the case (e.g. m is of type t_i), then the corresponding logic program p_i is extracted from RB and submitted to the Caller. The Caller then invokes the DLV system, asking for a model for $p_i \cup FB$. The resulting (best) model M is then converted in a burst of GSA$_2$ messages through the unwrapper UNWR, which takes advantage of an inverse translation table *TR2*.

Predicates stored in FB are classified as *temporary* (they persist on the fact base a finite amount of time), or *persistent* (they are stored permanently in FB, and persist from session to session). The Refresher module is in charge to periodically purge FB from temporary facts.

3.2 The WOLF framework

The WOLF (Web OntoLogy Framework) consists of a complete set of predicates intended to describe the Web and the user in a logic model. It is designed in order to comprehend as much as possible anything which the GSA programmer may want to deal with within the reasoning engine. Following the classic distinction of the planning field (see, e.g. [8]) predicates are divided in two main categories: fluents and actions. Fluents models the Web and user status, whereas actions are employed in order to indicate actions GSA$_2$ should perform, in response to a given event.

Fluents are intended in order to describe:

1. Knowledge Exchange: trust links among agents, physical reliability and knowledge of each agent.
2. Document Ontologies: topic taxonomies, relationships among concepts, documents and keywords.
3. Meta Search Modelling: reliability and responsiveness of each search engine. Quality of the search engine with respect to any topic.
4. User profiling: explicit and implicit feedback from the user; documents visited, time user has spent on them; Submitted queries, favorite search engines etc.

Reasoning is performed on some object types related each other: peers (remote GSA$_2$ instances), queries (set of keywords), documents (annotated lists of words and hyperlinks), bunch of documents (generic set of documents), topics/concepts (nested categories of documents), words.

For instance, WOLF includes predicates like `authority(p,o,c)`, stating that the authority (competence) of the peer p on the object o is given by the integer value c; or `up(p,t)`, which indicates the peer p was reported alive at time t. As for the web structure modeling the framework includes primitives like `link(u1,u2)`, which is true if GSA knows that the document $u1$ links $u2$. The existence of a query q is modelled by the fact `query(q)`, and `concept(c)` means that there exists the category of documents c. `contains(c,d)` states that the topic d is contained by the topic c.

Information retrieval interfacing is provided by facts like `document_score(u,q,s)`, meaning that the document u scores s with respect to the query q. `word_score(w,c,s)` express that the word w is related to the concept c with score s.

As for Meta Search Modelling, a fact like `retrieved(u,m,q,t)`, expresses that at time t, document u has been retrieved from the search-engine m when it has been queried with query q.

User activity is modelled as well, through predicates like `visited(u,u1,t)` that express the user has visited document u at time t, coming from document $u1$, or like `preferred(u,t)` which states that the document u has been added within the Favorites list of the user at time t.
In order to allow the Designer to fruitfully program knowledge sharing, each fluent fact is equipped with standard attributes comprehending the time the fact has been generated and an identification of the Agent which asserted the information. For the sake of readability, and unless confusion may arise, we will omit such fields whenever they are not necessary (for instance the fact `link(u1,u2)` is actually stored as `link(u1,u2,t,p)` where t is the time the fact were asserted and p is the asserting peer).

Agents may be local (e.g. the current modules IRis, MEry,PiPpi and loUIse), or remote (any peer from the GSA$_2$ network).

Actions can be divided in the following main categories:

1. Requests for knowledge exchange: requests for lists of reachable agents, requests for lists of experts on a given topic, requests for documents on a given topic.
2. Requests for document classification and mining: requests for evaluation and classification of documents, with respect to any given topic. Requests for significative keywords representative of some topic.
3. Requests for meta-search: requests for set of documents for any topic, requests for list of available search engines, etc.
4. Interaction with the user: requests to show a document, requests to present any kind of information to the user.

Most important action predicates are in the form `askforobject(o)`, where o is some kind of object (e.g. a document), or in the form `askforrelationship(o,o')`. In the former case some agent is prompted to answer with information about the object o, whereas in the latter case information about relationship between two objects is requested. For instance, the action `askforobject(q)`, where q is q query, if submitted to PiPpi, is intended in order to ask the whole GSA Network for the query q. `askforrelationship(q,p)` asks only the peer p instead.
Many actions concern document classification and mining: for instance,

– `askforrelationship(u,q)`, where u is a document and q is a query, is answered by IRis, with a fact `document_score(u,q,s)`.
– `askforrelationship(u,c)`, where u is a document and c a concept, asks an agent about information on how the document u could be classified inside the class of documents c.
– `request_induction(c,s)` (which is an alias to `askforrelationship(c,s)`), is employed to ask IRis in order to try to induce the concept c from the bunch of documents s.

As for meta-searching, we may cite, for instance, `askengineforquery(m,q)` (same as
`askforrelationship(m,q)`, which asks meta-engine m for the query q).
Some actions that are usually directed to the user interface (the loUIse agent) appear like, e.g., `suggest_document(u)`, which prompts the user interface to suggest the document u or `suggest_query(q)`, which prompts the user interface to suggest the (possibly refined) query q.

3.3 The KatheRinE lifecycle

We are now ready to present an example of the KatheRinE activity and some example of built-in logic program the developer is able to design. Assume a message m, of the kind *QueryRequest* enters KatheRinE. This kind of message may come either from loUIse (whenever the user submits a query in a suitable text field), or from some GSA$_2$ peer around the GSA$_2$ network, asking for answers to a text query. Let m contain the query $q = \{$ “search”,“engine” $\}$. m is wrapped to a set of logical facts. In this case new facts asserted on the internal fact base are:

```
    query(qx00000000,20030402163056,"local.louise").
    belongs("search",qx0000,20030402163056,1,"local.louise").
    belongs("engine",qx0000,20030402163056,2,"local.louise").
```

Facts above tells KatheRinE that q exists from April 2th, 2003, 16:30:56 UT, and that this information has been asserted by the local agent loUIse. Words belonging to q are asserted as well. If *QueryRequest* belongs to the trigger table, and we assume this is the case, KatheRinE is prompted to perform some additional operation. In order to track trigger events, these are labelled and associated with an order number, where 0 corresponds to the last trigger event. The fact `trigger(qx0000,"local.louise",0)` is thus asserted. A logic program associated to the *QueryRequest* message type is then invoked. Such program describes which action to take in this case. For instance, we may want to answer by consulting search-engines, or taking advantage of the GSA$_2$ network. Such a program sounds like the following:

```
1. t(Q,X) :- trigger(Q,X,0), query(Q).
2. suggest_object(U,X) :-
    t(Q,X),
    document_score(U,Q,S,_,_),
    S > 500.
3. askforobject(Q,"local.meri") :- t(Q,"local.louise").
4. askforobject(Q,"local.pippi") :- t(Q,"local.louise").
5. askforobject(U2,"local.iris") :-
    t(Q,"local.louise"),
    suggest_object(U1,_), link(U1,U2,_),
    not document_score(U2,Q,_,_,_).
```

The purpose of the above program is to take advantage of any resource in order to answer the user request. Only the local fact base is consulted if the same request did not come from the loUIse agent. In particular, rule (1) sends out to the user interface any document already scored with respect the same query. Rule (3) and (4) activate MEry (the meta-search module) and PiPpi (the peer to peer network) respectively, only if the trigger event came from “`local.louise`”, i.e. the user interface. Rule (5) ask the IRis agent for an anticipated scoring of documents linked from already suggested documents. The evaluation of the above program produces an unique model M. M is

then unwrapped: in particular any action predicate occurring in it, is converted in a corresponding message. For instance, the fact
`askforobject(qx0000,"local.pippi")` is converted to a corresponding message *AskForObject*, containing the requested query and directed to the PiPpi agent, whereas `suggest_object` predicates are wrapped to messages usually directed to the user interface.

This way, it is possible to program search strategies taking full advantage of the logic programming declarativity. A few examples follow.

3.4 Managing Persistence of Interest

Profiling the user from her activity is an important issue. In general the GSA_2 Agent should continue autonomously its search process while the user is browsing. Anyway, it is very important to detect *persistence of interest*, i.e. to automatically understand whether the user has changed her interest about a given topic during the browsing process. This is, e.g. the aim to be achieved by assisting browsers such as Letizia [21]. The high power of the internal ASP-based language allows us to be very flexible to this respect, and to program very easily strategies for detecting changes of interest, as the following example shows.

We assume a query q has been submitted in the past from the users. Usually KatheRinE reacts to queries asking one or more modules for relevant documents related to q. External modules keep on sending documents depending on their internal strategy. It is worth to note that many events may have happened meanwhile, indicating a change of interest. Thus, suggested objects (i.e. those u such that `suggestedObject(u,q)` is true for some q) coming from external agents are redirected to the user only in case the user interest is still persistent. We assume the following program is associated to a message of the kind *SuggestObjects*, usually containing a bunch of suggested objects.

```
1. t(Type, Object, Agent) :-
     trigger(Type, Object, Agent, Position).
2. suggestObject(U, Agent) :- t("UserQuery", Query, Agent),
     stillActive(Query), suggestedObject(U, Query).
3. stillActive(Query) :- query(Query, Time, _),
     not contextChanged(Event, Time).
4. contextChanged(Query, Time) :- query(Query1, Time),
     query(Query, Time1), Time1 > Time.
```

The meaning of this short piece of code is quite simple. Rule (1) considers all interesting events (triggers). This rule can be changed in order to select past trigger events worth to process with respect to the persistence of interest. Rule (2) expresses the chosen interest policy: if some module retrieved any object meaningful for a given query, it has to be notified to the subject that caused the trigger to fire, unless context changes have arisen (rule (3)).

A *context change* can be interpreted in many ways, and it is worth to note that it can be easily described through just few rules. In this case, we wanted just to skip all notifications if the user asked for a new query *after* the one the module should be answering to. Considering other user events, we can enrich or modify this behavior: it is not so difficult to imagine how to catch a context change due to some other user activity, for instance through measuring time spent by the user on suggested pages, etc.

3.5 Selecting useful set of peers to be queried

One important issue regarding peer-to-peer networks is to ensure data persistency (at least one alive peer must hold a given information), freshness (information must be up to date as much as possible) and consistency (it should not exists two peers asserting different things about the same object, and, if this is the case, data must be reconciled). Such issues are solved through different approaches (see e.g. [24]).

In many cases, it is necessary to limit the number of peers to be consulted for a given topic. At the same time peers which were recently reachable are preferable with respect to elder peers. If the same information is stored at several sites, a lot of time could be saved by querying only a restricted set. The following program suggest how to implement a peer selection strategy based on the reduction of overlapping data, and on the selection of freshest peers.

We assume q is a query, and we want to ask the GSA$_2$ network for this query. Anyway, we prefer not to ask the whole network, but only the minimal set which guarantees the whole set of documents spread throughout the network is covered. In case the same document is available at different sites, we prefer to query the freshest one. The local GSA$_2$ instance is aware of documents covered by each peer through facts of the kind `document_score(u,_,_,_,p)`. Such a fact means that the peer p has sometimes scored the document u, and presumably, it has indexed this document. The fact `currenttime(t)` returns the current UT time, whereas a fact `up(p,t)` tells that the peer p has been reported alive at time t. A peer selection program could look like the following:

```
1. t(Q) :- trigger(Q,_,0).
2. candidate(P,T) :- currenttime(T1),
                     up(P,T2),
                     T = T1-T2,
                     document_score(U,_,_,_,P).
3. tobecovered(U) :- document_score(U,_,_,_,_).
4. incover(P) v outofcover(P) :- candidate(P,T).

5. covered(U) :- tobecovered(U),
                 incover(P),
                 document_score(U,_,_,_,P).
6. :- document(U), not covered(U).
7. :˜ incover(P),candidate(P,T).  [T:]
8. askforobject(Q,P) :- t(Q), incover(P).
```

The above program works as follows: we generate a table of candidate peers (rule 2) by means of the *candidate* predicate. `candidate(p,t)` will mean that the peer p has been reported alive t seconds ago. Then, through rule 4, we take advantage of disjunction in order to specify a search space where a candidate peer may be, or may be not, part of the selected cover (`incover(p)` will mean that p is part of the set to be queried). Through rule 6, we filter out those subsets of candidates not covering every document. The weak constraint of Line 7 weights each peer by its alive time, preferring those peers with smaller alive time. Rule 8 performs the wanted action for those peers belonging to the cover.

3.6 Trusted peers

Another issue arising in peer to peer systems is strictly tied to the problem of trust. Peers may store aged and/or incorrect information at their site. Thus, each agent should be in charge to choose its policy on trustworthiness.

In [11], the authors propose an approach based on the definition of a semantic policy language. Each peer can choose a personal policy on trustworthiness; a client peer is trusted on the basis of the policy of the server peer.
We show next how to implement a possible trustworthiness policy. Each peer is coupled with a plausibility value, comparing data about documents present within the local fact base (L) and in a given peer fact base (P). The rationale is that as much facts asserted by a remote peer p can be locally verifiable, as much p can be trusted.

In this case, we assume data a peer has to be trusted about is the 'document matching degree', i.e. the degree of pertinence of a given document with respect to a given topic.

Let t be a given document category, and u a document, then we denote as $S_L(u,t)$ the score (matching degree) of u with respect the category t, stored in L, whereas $S_P(u,t)$ will be the same value as stored in P. Let $H = D(L) \cap D(P)$, where $D(L)$ is the document set stored within the local fact base and $D(P)$ the document set stored in peer P, then:

$$Pl_P(t) = \frac{|\{u \in H : S_L(u,t) = S_P(u,t)\}|}{|H|}$$

is the plausibility index of the fact base P with respect to the topic t. Pl_P is given by the ratio among the number of documents, stored with same score value, belonging to L and P, and those that are both in P and in L (regardless of their score). The following example creates a set of peers estimating trustworthiness and selecting peers where plausibility exceeds a given threshold.

```
1. trusting_threshold(90).
2. normalize(P,N) :- peer(P), #count{U: document-score(U,_,S,_,"local.louise"),
                document_score(U,_,S,_,P)} = M, N = M*100.
3. plausibility(P,R) :- normalize(P,N),
                        #count{U: document(U,"local.louise"),
                        document(U,P)} = M, div(N,M,R).
4. trusted(P) :- plausibility(P,R), trusting_threshold(T), R > T.
```

The meaning of the above program is the following: through rule (1), we set a trusting threshold, then, by means of rules (2) and (3), we compute the plausibility value for each peer (`plausibility(p,r)` will mean that the plausibility of the peer p is r). Finally, rule (4) selects the set of trusted peers (a peer p is trusted if `trusted(p)` is true). The set of trusted peers can then be employed in any way the system designer may want, e.g. for querying information.

4 Conclusions

In this paper we have described the new architecture of the GSA$_2$ prototype, and, in particular, how an Agent reasoning on the basis of a logic programming framework can be exploited to control the whole multi-agent system. The internal reasoning formalism allows to easily specify different behaviors of the system, and to quickly implement any Meta-search, Information Retrieval and Information Exchange strategy. We are currently working toward the complete integration of each module into the final GSA$_2$ release. The next step of our ongoing research is to extend the KatheRinE architecture in a more general fashion, in order to provide a more extended framework where ASP can be exploited within multi-agent applications of any nature.

References

1. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2002. In press.
2. F. Buccafurri, N. Leone, and P. Rullo. Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.
3. A. Castellucci, G. Ianni, D. Vasile, and S. Costa. Surfing and searching the web using a semi-adaptive meta-engine. In *International Conference on Information Technology: Coding and Computing (ITCC). Las Vegas, Nevada (USA)*, pages 416–420, 2-4 Aprile 2001.
4. E. Damiani, B. Oliboni, E. Quintarelli, and L. Tanca. Modeling users' navigation history. *Seventeenth International Joint Conference on Artificial Intelligence*, 2001.
5. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI) 2003*, Acapulco, Mexico, August 2003. Morgan Kaufmann Publishers.
6. T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The DLV System. In Jack Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC*, College Park, Maryland, June 1999. Computer Science Department, University of Maryland. Workshop Notes.
7. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.
8. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Using the `dlv` system for planning and diagnostic reasoning. In *Proceedings of the 14th Workshop on Logic Programming (WLP'99)*, pages 125–134. GMD – Forschungszentrum Informationstechnik GmbH, Berlin, January 2000. ISSN 1435-2702.
9. Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. Progress Report on the Disjunctive Deductive Database System `dlv`. In Troels Andreasen, Henning Christiansen, and Henrik Legind Larsen, editors, *Proceedings International Conference on Flexible Query Answering Systems (FQAS'98)*, number 1495 in Lecture Notes in AI (LNAI), pages 148–163, Roskilde University, Denmark, May 1998. Springer.
10. F. Bellifemine and A. Poggi and G. Rimassa. Developing Multi-agent Systems with a FIPA compliant Agent Framework. *Software - Practice and Experience*, 31:103–128, 2001.
11. T. Finin and A. Joshi. Agents, trust, and information access on the semantic web. *ACM SIGMOD Record*, 31:30–35, 2002.
12. FIPA (Fundation for Information Physical Agents) web site. `http://www.fipa.org`.
13. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.
14. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
15. N. Green, P. G. Ipeirotis, and L. Gravano. SDLIP + STARTS = SDARTS a protocol and toolkit for metasearching. In *ACM/IEEE Joint Conference on Digital Libraries*, pages 207–214, 2001.
16. G. Ianni. Intelligent anticipated exploration of web sites. *AI Communications*, 14(4):197–214, 2001.
17. P. G. Ipeirotis and L. Gravano. Distributed search over the hidden web: Hierarchical database samplingand selection. *28th International Conference on Very Large Data Bases (VLDB 2002)*, 2002.

18. D. Kazakov and D. Kudenko. Machine learning and inductive logic programming for multi-agent systems. *9th ECCAI Advanced Course ACAI 2001 and Agent Link's 3rd European Agent Systems Summer School*, LNCS 2086:246–270, 2001.
19. J. U. Kietz, R. Volz, and A. Maedche. Extracting a domain-specific ontology from a corporate intranet. In Claire Cardie, Walter Daelemans, Claire Nédellec, and Erik Tjong Kim Sang, editors, *Proceedings of the 4th Conference on Computational Natural Language Learning and of the 2nd Learning Language in Logic Workshop, Lisbon, 2000*, pages 167–175. Association for Computational Linguistics, Somerset, New Jersey, 2000.
20. A. Kobsa, J. Koenemann, and W. Pohl. Personalized hypermedia presentation techniques for improving online customer relationships, 2001.
21. H. Lieberman. Letizia: An agent that assists web browsing. *Proc. of the 14th Int. Joint Conf. on Artificial Intelligence, IJCAI 95, Montréal, Québec, Canada*, pages 924–929, 1995.
22. A. Maedche and S. Staab. Learning ontologies for the semantic web. volume 16, pages 72–79, 2001.
23. T. Matsui, N. Inuzuka, and H. Seki. A Proposal for Inductive Lerning Agent Using First-Order Logic. In *Tenth International Conference on Inductive Logic Programming (ILP2000)*, July 2000.
24. W. Nejdl, B. Wolf, et al. Edutella: A P2P network infrastructure based on RDF. In *11th International WWW conference*, May 2002.
25. Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A-Prolog Decision Support System for the Space Shuttle. In Gopal Gupta, editor, *Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, number 1551 in Lecture Notes in Computer Science, pages 169–183. Springer, 1999.
26. Gerald Pfeifer. Tutorial: Answer Set Programming. 8th European Conference on Artificial Intelligence (JELIA), Cosenza, Italy, September 2002.
27. F. Sadri, F. Toni, and P. Torroni. Logic agents, dialogues and negotiation: An abductive approach. *Proceedings of the Symposium on Information Agents for E-Commerce, AISB'01, York, UK*, March 2001.
28. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, June 2002.
29. T. C. Son, C. Baral, and S. McIlraith. Planning with Different Forms of Domain-Dependent Control Knowledge – An Answer Set Programming Approach. In Proceedings of, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, LNAI 2173*, pages 226–239. Springer Verlag, September 2001.